

Ghid complet: Interfețe funcționale în Java

Ce este o interfață funcțională? (Definiția exactă)

O **interfață funcțională** este o interfață care satisface această condiție:

Numărul de metode abstracte = EXACT 1

Reguli importante:

- ✓ Poate avea oricâte metode `default`
- ✓ Poate avea oricâte metode `static`
- ✓ Poate avea metode din `Object` (`toString`, `equals`, `hashCode`)
- ✗ NU poate avea 0 metode abstracte
- ✗ NU poate avea 2+ metode abstracte

Algoritm de analiză pas cu pas

Pasul 1: Fac inventarul pentru fiecare interfață

Pentru interfața X:

1. Metode abstracte proprii = ?
2. Metode abstracte moștenite = ?
3. Metode default care suprascriu abstracte = ?
4. TOTAL metode abstracte = (1 + 2 - 3)

Pasul 2: Urmăresc moștenirea

java

// Exemplu simplu

```
interface A { void metoda1(); }           // 1 abstractă
interface B extends A { void metoda2(); } // 1 + 1 = 2 abstracte
interface C extends A {
    default void metoda1() { ... }        // 1 - 1 = 0 abstracte
}
```

Trucuri și capcane comune

🎯 **Truc 1: Metodele default "anulează" abstractele**

java

```
interface Base { void f(); } // 1 abstractă
interface Child extends Base {
    default void f() { ... } // 0 abstracte! (f nu mai e abstractă)
}
```

🎯 Truc 2: Moștenirea multiplă - aceleași metode se "îmbină"

java

```
interface A { void metoda(); }
interface B { void metoda(); } // Aceeași semnătură!
interface C extends A, B {} // Tot 1 abstractă (nu 2!)
```

🎯 Truc 3: Moștenirea multiplă - metode diferite se "adună"

java

```
interface A { void metoda1(); }
interface B { void metoda2(); } // Semnături diferite!
interface C extends A, B {} // 2 abstracte = NU funcțională
```

🎯 Truc 4: Metodele din Object nu contează

java

```
interface Test {
    void metoda(); // 1 abstractă
    String toString(); // NU contează! (din Object)
    boolean equals(Object o); // NU contează! (din Object)
}
// Rezultat: 1 abstractă = funcțională ✅
```

🎯 Truc 5: Suprascrierile de metode default

java

```
interface A {
    void abstracta();
    default void cu_implementare() { ... }
}
interface B extends A {
    default void abstracta() { ... }    // Suprascrie abstracta!
    void cu_implementare();            // Face default-ul abstract!
}
// A: 1 abstractă, B: 1 abstractă (diferită!)
```

Cazuri speciale și capcane

⚠ Capcana 1: Interfața goală

java

```
interface Goala {}                                // 0 abstracte = NU funcțională
```

⚠ Capcana 2: Doar metode default

java

```
interface NumarDefault {
    default void metoda1() { ... }
    default void metoda2() { ... }
}
// 0 abstracte = NU funcțională
```

⚠ Capcana 3: Metode cu aceeași numele, parametri diferiți

java

```
interface Overload {
    void metoda(int x);                // 1 abstractă
    void metoda(String x);             // 2 abstracte total
}
// NU funcțională
```

⚠ Capcana 4: Generic vs Non-generic



java

```
interface A { void metoda(Object x); }
interface B { void metoda(String x); }    // Semnături diferite!
interface C extends A, B {}              // 2 abstracte = NU funcțională
```

Exemple practice cu analiza completă


Exemplu 1: Moștenire simplă

java

```
interface Functie_1 { int f(double x); }  
// Analiza: 1 metodă abstractă proprie → Funcțională   
  
interface Functie_2 extends Functie_1 {  
    default void afisare(int x) { ... }  
}  
  
// Analiza:  
// - Moștenește: f(double x) abstractă  
// - Adaugă: afisare default (nu contează)  
// - Total: 1 abstractă → Funcțională 
```


Exemplu 2: Default care suprascrie

java

```
interface Functie_3 extends Functie_2 {  
    default int f(double x) { return (int) (x + 1); }  
}  
  
// Analiza:  
// - Moștenește: f(double x) abstractă  
// - Suprascrie cu default: f(double x) nu mai e abstractă  
// - Total: 0 abstracte → NU funcțională 
```

Exemplu 3: Moștenire multiplă cu aceeași metodă

java

```
interface Functie_4 extends Functie_1, Functie_2 {}  
  
// Analiza:  
// - De La Functie_1: f(double x) abstractă  
// - De La Functie_2: f(double x) abstractă (aceeași!)  
// - Total: 1 abstractă → Funcțională 
```

Exemplu 4: Adăugare metodă nouă

java

```
interface Functie_5 extends Functie_2 { double g(int x); }  
// Analiza:  
// - Moștenește: f(double x) abstractă  
// - Adaugă: g(int x) abstractă  
// - Total: 2 abstracte → NU funcțională ❌
```

Checklist rapid pentru examen

✅ Întrebări cheie:

1. Câte metode abstracte are interfața?

- Ignoră default, static, și metodele din Object
- Ține cont de suprascrierea cu default

2. Ce moștenește?

- Copiază toate metodele abstracte din părinte
- Verifică dacă sunt aceleași metode în cazul moștenirii multiple

3. Ce adaugă nou?

- Metode abstracte noi se adună la total
- Metode default nu afectează totalul

4. Rezultatul final:

- 1 abstractă = Funcțională ✅
- Altceva = NU funcțională ❌

🚀 Trucuri pentru viteză:

- **Start de la interfața de bază** și urmărește în jos
- **Desenează un arbore** pentru moșteniri complexe
- **Marchează cu X** metodele care devin default
- **Numără doar la sfârșit** - nu te grăbi să numeri pe parcurs

De ce contează în practică?

Interfețele funcționale permit:

java

// Lambda expressions

Funcție_1 f1 = x -> (int) x * 2;

// Method references

Funcție_1 f2 = Math::round;

// Folosire în Stream API

list.stream().map(x -> x + 1).collect(toList());

Concluzie: Înțelegerea interfețelor funcționale este esențială pentru Java modern și programarea funcțională!